

MSDS Capstone – FA 24 DSCI 6051-02

# REPORT :

# AI for Solving New York Times Connection Game



**FALL - 2024**

**CONTENTS**

|  |  |  |
| --- | --- | --- |
| **Sl. No** | **Content** | **Pages** |
| **1.** | **AI for Solving New York Times Connection Game** | **3** |
| **2.** | **Objective** | **4** |
| **3.** | **Scope** | **5** |
| **4.** | **Business Understanding** | **6** |
| **5.** | **Data Understanding & Preparation** | **7** |
| **6.** | **Modeling** | **8** |
| **7.** | **Evaluation** | **9** |
| **8.** | **Source Code** | **10** |
| **9.** | **Source Code for API** | **15** |
| **10.** | **Out Put** | **20** |
| **11.** | **Web Interface: Word Group Generator** | **21** |
| **12.** | **Word Embedding Visualization** | **22** |
| **13.** | **Future Recommendations** | **23** |

**AI for Solving New York Times Connection Game**

**Introduction**

Connections is a word puzzle developed and published by The New York Times as part of The New York Times Games. It was released on June 12, 2023, during its beta phase. It is the second-most-played game that is published by the Times, behind [Wordle](https://en.wikipedia.org/wiki/Wordle).

## Team Members:

**Lakshmi Prasanna Kalluri** (lkall1@unh.newhaven.edu)

**Allupati, Jagannath Patro** (jallu1@unh.newhaven.edu )

**Dumpala, Srikanth** (sdump2@unh.newhaven.edu)

**Angadi, Chaithra**

(canga1@unh.newhaven.edu )

**Objective**

The primary objective of this project is to develop an AI-powered solution for the New York Times Connection Game by generating logical word groups and evaluating the AI-generated puzzles against the original puzzles to ensure accuracy and effectiveness.

**Key Problem:**

The New York Times Connection Game requires efficient and accurate word grouping based on semantic relationships, which can be challenging to automate. Our project aims to address this issue by leveraging AI to ensure consistent and logical groupings that resemble the human comprehension.

**Scope**

The project focuses on creating a Minimum Viable Product (MVP) that integrates: Automated grouping of words for a given puzzle using Word2Vec and similarity measures. A web-based interface for visualizing and interacting with grouped words. Evaluation metrics to assess the accuracy and effectiveness of the generated groups.

**Business  
Understanding**

**Business Objective:** Enhance the NYT Connection Game by improving grouping accuracy and user engagement.

**Success Criteria:**

* **Business:** Reliable and consistent word groupings that improve gameplay experience.
* **Technical:** High success rates and accurate groupings that align with human-like understanding.

**Planning**

**Tools:** Word2Vec, Flask, Python (Gensim, NumPy, Plotly).

**Steps:** Data preparation, model development, evaluation, and deployment of a web-based interface for visualization and interaction.

**Data Understanding & Preparation**

* **Data Sources: Puzzles from the New York Times Connection Game.**
* **Source: [NYT Puzzles JSON](**[**https://anonymous.4open.science/r/making-new-connections-78D1/nyt\_puzzles.json**](https://anonymous.4open.science/r/making-new-connections-78D1/nyt_puzzles.json)**)**
* **Integrated word embeddings using Word2Vec to represent words in a semantic vector space.**
* **Some words were not present in the Word2Vec vocabulary. This was addressed by filtering out such words or substituting with similar ones.**
* **Word embeddings are generated for all valid words to facilitate similarity calculations.**

**Modeling**

* **The primary model used was Word2Vec, a neural embedding model, to generate vector representations of words and calculate semantic similarities.**
* **The Word2Vec model was trained using preprocessed word datasets.**
* **Cosine similarity was calculated between word vectors to group semantically related words.**
* **Start by forming 8 most similar word pairs using Word2Vec and cosine similarity.**
* **Further combine these pairs into 4 final groups based on average similarity metrics.**

**Evaluation**

* **Groups generated based on their similarities are evaluated through metrics like success rate and overlap with original groups.**
* **The model demonstrated a significant success rate in replicating human-like groupings, achieving the desired standards of accuracy and logical consistency.**

**Source code**

import json

from collections import Counter

import gensim

from gensim.models import Word2Vec

from sklearn.metrics.pairwise import cosine\_similarity

import numpy as np

def train\_word2vec\_model(words, vector\_size=100):

"""Trains a Word2Vec model on a list of words.

Args:

words: A list of words.

vector\_size: The dimensionality of the word vectors.

Returns:

A trained Word2Vec model.

"""

model = Word2Vec([words], min\_count=1, vector\_size=vector\_size)

return model

# Function to filter words present in the embedding model

def filter\_words(words, model):

# Filter words to ensure they are present in the model's vocabulary

return [word for word in words if word in model.wv.key\_to\_index]

def load\_puzzle\_data(filename):

with open(filename, 'r') as f:

return json.load(f)

def extract\_word\_pairs(words):

pairs = []

for i in range(len(words)):

for j in range(i+1, len(words)):

pairs.append((words[i], words[j]))

return pairs

def calculate\_cosine\_similarity(word\_vectors, word\_pairs):

similarities = []

for word1, word2 in word\_pairs:

vector1 = word\_vectors[word1]

vector2 = word\_vectors[word2]

similarity = np.dot(vector1, vector2) / (np.linalg.norm(vector1) \* np.linalg.norm(vector2))

similarities.append((similarity, word1, word2))

return sorted(similarities, reverse=True) # Sort by similarity in descending order

def form\_8\_unique\_pairs(similarities):

used\_words = set()

pairs = []

for similarity, word1, word2 in similarities:

if word1 not in used\_words and word2 not in used\_words:

pairs.append((word1, word2))

used\_words.add(word1)

used\_words.add(word2)

if len(pairs) == 8:

break

return pairs

def group\_word\_pairs(pairs, similarity\_matrix, words):

groups = []

available\_pairs = list(range(len(pairs)))

while len(available\_pairs) >= 2:

max\_similarity = -1

best\_pair\_indices = None

for i in available\_pairs:

for j in available\_pairs:

if i != j:

pair1, pair2 = pairs[i], pairs[j]

avg\_similarity = np.mean([

similarity\_matrix[words.index(w1), words.index(w2)]

for w1 in pair1 for w2 in pair2

])

if avg\_similarity > max\_similarity:

max\_similarity = avg\_similarity

best\_pair\_indices = (i, j)

if best\_pair\_indices:

i, j = best\_pair\_indices

groups.append(pairs[i] + pairs[j])

available\_pairs.remove(i)

available\_pairs.remove(j)

return groups

def compute\_success\_rate(original\_groups, generated\_groups):

matched\_groups = 0

for generated\_group in generated\_groups:

# Check if the generated group exactly matches any original group

for original\_group in original\_groups:

if set(generated\_group) == set(original\_group):

matched\_groups += 1

break # Stop checking once a match is found

total\_groups = len(original\_groups)

success\_rate = (matched\_groups / total\_groups) \* 100

return success\_rate

def compute\_partial\_success\_rate(original\_groups, generated\_groups):

total\_words = sum(len(group) for group in original\_groups) # Total words across all original groups

matched\_words = 0

# For each generated group, count overlaps with original groups

for generated\_group in generated\_groups:

for original\_group in original\_groups:

# Count words in the generated group that overlap with the current original group

matched\_words += len(set(generated\_group) & set(original\_group))

# Calculate the success rate as a percentage

success\_rate = (matched\_words / total\_words)

return success\_rate

def compute\_success\_rate\_min\_two(original\_groups, generated\_groups):

matched\_groups = 0

for generated\_group in generated\_groups:

for original\_group in original\_groups:

# Count words in the generated group that overlap with the current original group

overlap = len(set(generated\_group) & set(original\_group))

if overlap >= 2: # At least two words match

matched\_groups += 1

break # Stop checking once a match is found

total\_groups = len(original\_groups)

success\_rate = (matched\_groups / total\_groups) \* 100

return success\_rate

def calculate\_success\_rate(original\_groups, generated\_groups):

"""

Calculate the success rate of grouping words based on the overlap between original and generated groups.

:param original\_groups: List of original groups (list of lists)

:param generated\_groups: List of generated groups (list of lists)

:return: Success rate as a percentage

"""

total\_words = sum(len(group) for group in original\_groups)

correctly\_grouped = 0

# Check matches between original and generated groups

for original\_group in original\_groups:

max\_overlap = 0

for generated\_group in generated\_groups:

# Count overlapping words

overlap = len(set(original\_group) & set(generated\_group))

max\_overlap = max(max\_overlap, overlap)

correctly\_grouped += max\_overlap

# Calculate success rate

success\_rate = (correctly\_grouped / total\_words) \* 100

return success\_rate

def main():

puzzle\_data = load\_puzzle\_data("/content/sample\_data/nyt\_puzzles.json")

for puzzle\_index, puzzle in enumerate(puzzle\_data): # Use enumerate to get index

words = []

original\_groups = []

for category in puzzle:

original\_groups.append(category['words']) # Store the original groups

words.extend(category['words'])

# Train the Word2Vec model

model = train\_word2vec\_model(words)

# Filter words to ensure they are present in the model's vocabulary

filtered\_words = filter\_words(words, model)

if not filtered\_words:

print(f"Puzzle {puzzle\_index + 1}: No valid words found.")

continue

# Extract word pairs

word\_pairs = extract\_word\_pairs(filtered\_words)

# Create a dictionary of word vectors for easy access

word\_vectors = {word: model.wv[word] for word in filtered\_words}

# Calculate cosine similarities

similarities = calculate\_cosine\_similarity(word\_vectors, word\_pairs)

if not similarities:

print(f"Puzzle {puzzle\_index + 1}: No similarities calculated.")

continue

# Form top 8 unique pairs

top\_8\_pairs = form\_8\_unique\_pairs(similarities)

# Prepare similarity matrix for grouping

similarity\_matrix = np.zeros((len(filtered\_words), len(filtered\_words)))

for i, word1 in enumerate(filtered\_words):

for j, word2 in enumerate(filtered\_words):

similarity\_matrix[i, j] = np.dot(

word\_vectors[word1], word\_vectors[word2]

) / (np.linalg.norm(word\_vectors[word1]) \* np.linalg.norm(word\_vectors[word2]))

# Group word pairs into groups

grouped\_pairs = group\_word\_pairs(top\_8\_pairs, similarity\_matrix, filtered\_words)

# Compute success rate based on at least two matching words

success\_rate = calculate\_success\_rate(original\_groups, grouped\_pairs)

# Print the results

print(f"\nPuzzle {puzzle\_index + 1}: Original Groups:")

for group\_index, group in enumerate(original\_groups):

print(f"Original Group {group\_index + 1}: {group}")

print(f"\nPuzzle {puzzle\_index + 1}: Top 8 Most Similar Word Pairs:")

for word1, word2 in top\_8\_pairs:

print(f"{word1} - {word2}")

print("\nGenerated Groups:")

for group\_index, group in enumerate(grouped\_pairs):

print(f"Generated Group {group\_index + 1}: {group}")

print(f"\nPuzzle {puzzle\_index + 1}: Success Rate: {success\_rate:.2f}%")

if \_\_name\_\_ == "\_\_main\_\_":

main()

**Source code for API**

from flask import Flask, request, render\_template

from gensim.models import Word2Vec

from sklearn.decomposition import PCA

import numpy as np

from sklearn.metrics.pairwise import cosine\_similarity

import plotly.express as px

import pandas as pd

app = Flask(\_\_name\_\_)

# Core Functions

def train\_word2vec\_model(words, vector\_size=100):

"""Trains a Word2Vec model on a list of words."""

model = Word2Vec([words], min\_count=1, vector\_size=vector\_size)

return model

def filter\_words(words, model):

"""Filters words to ensure they are present in the model's vocabulary."""

return [word for word in words if word in model.wv.key\_to\_index]

def extract\_word\_pairs(words):

"""Extracts all pairs of words."""

pairs = []

for i in range(len(words)):

for j in range(i + 1, len(words)):

pairs.append((words[i], words[j]))

return pairs

def calculate\_cosine\_similarity(word\_vectors, word\_pairs):

"""Calculates cosine similarity between word pairs."""

similarities = []

for word1, word2 in word\_pairs:

vector1 = word\_vectors[word1]

vector2 = word\_vectors[word2]

similarity = np.dot(vector1, vector2) / (np.linalg.norm(vector1) \* np.linalg.norm(vector2))

similarities.append((similarity, word1, word2))

return sorted(similarities, reverse=True) # Sort by similarity in descending order

def form\_8\_unique\_pairs(similarities):

"""Forms the top 8 unique word pairs based on similarity."""

used\_words = set()

pairs = []

for similarity, word1, word2 in similarities:

if word1 not in used\_words and word2 not in used\_words:

pairs.append((word1, word2))

used\_words.add(word1)

used\_words.add(word2)

if len(pairs) == 8:

break

return pairs

def group\_word\_pairs(pairs, similarity\_matrix, words):

"""Groups word pairs into groups based on similarity matrix."""

groups = []

available\_pairs = list(range(len(pairs)))

while len(available\_pairs) >= 2:

max\_similarity = -1

best\_pair\_indices = None

for i in available\_pairs:

for j in available\_pairs:

if i != j:

pair1, pair2 = pairs[i], pairs[j]

avg\_similarity = np.mean([

similarity\_matrix[words.index(w1), words.index(w2)]

for w1 in pair1 for w2 in pair2

])

if avg\_similarity > max\_similarity:

max\_similarity = avg\_similarity

best\_pair\_indices = (i, j)

if best\_pair\_indices:

i, j = best\_pair\_indices

groups.append(pairs[i] + pairs[j])

available\_pairs.remove(i)

available\_pairs.remove(j)

return groups

def compute\_success\_rate(original\_groups, generated\_groups):

"""Computes the success rate between original and generated groups."""

matched\_groups = 0

for generated\_group in generated\_groups:

# Check if the generated group exactly matches any original group

for original\_group in original\_groups:

if set(generated\_group) == set(original\_group):

matched\_groups += 1

break # Stop checking once a match is found

total\_groups = len(original\_groups)

success\_rate = (matched\_groups / total\_groups) \* 100 if total\_groups else 0

return success\_rate

def calculate\_success\_rate(original\_groups, generated\_groups):

"""

Calculate the success rate of grouping words based on the overlap between original and generated groups.

:param original\_groups: List of original groups (list of lists)

:param generated\_groups: List of generated groups (list of lists)

:return: Success rate as a percentage

"""

total\_words = sum(len(group) for group in original\_groups)

correctly\_grouped = 0

# Check matches between original and generated groups

for original\_group in original\_groups:

max\_overlap = 0

for generated\_group in generated\_groups:

# Count overlapping words

overlap = len(set(original\_group) & set(generated\_group))

max\_overlap = max(max\_overlap, overlap)

correctly\_grouped += max\_overlap

# Calculate success rate

success\_rate = (correctly\_grouped / total\_words) \* 100

return success\_rate

def generate\_pca\_plot(model, filtered\_words, groups):

"""Generates a PCA plot for word embeddings."""

word\_vectors = [model.wv[word] for word in filtered\_words]

# Perform PCA

pca = PCA(n\_components=2)

reduced\_vectors = pca.fit\_transform(word\_vectors)

# Create a DataFrame for Plotly

df = pd.DataFrame(reduced\_vectors, columns=["PCA Component 1", "PCA Component 2"])

df['Word'] = filtered\_words

# Assign colors to words based on their group

word\_to\_group = {}

group\_colors = ['red', 'green', 'blue', 'orange', 'purple', 'yellow', 'cyan', 'magenta'] # Adjust as necessary

for i, group in enumerate(groups):

for word in group:

word\_to\_group[word] = group\_colors[i % len(group\_colors)]

df['Color'] = df['Word'].apply(lambda word: word\_to\_group.get(word, 'black'))

# Create interactive PCA plot using Plotly

fig = px.scatter(df, x="PCA Component 1", y="PCA Component 2", text="Word", color="Color",

title="Word Embedding Visualization (PCA)",

labels={"PCA Component 1": "PCA Component 1", "PCA Component 2": "PCA Component 2"})

fig.update\_traces(marker=dict(size=12),

textposition='top center',

showlegend=False)

# Convert to HTML for embedding in Flask

return fig.to\_html(full\_html=False)

@app.route("/", methods=["GET", "POST"])

def index():

if request.method == "POST":

words = request.form["words"].split(",")

original\_groups\_input = request.form["original\_groups"]

# Convert original groups input into list of lists

original\_groups = [group.split(",") for group in original\_groups\_input.split(";")]

# Train Word2Vec model

model = train\_word2vec\_model(words)

# Filter words to ensure they are present in the model's vocabulary

filtered\_words = filter\_words(words, model)

if not filtered\_words:

return render\_template("index.html", error="No valid words found in the model's vocabulary.")

# Extract word pairs

word\_pairs = extract\_word\_pairs(filtered\_words)

# Create a dictionary of word vectors for easy access

word\_vectors = {word: model.wv[word] for word in filtered\_words}

# Calculate cosine similarities

similarities = calculate\_cosine\_similarity(word\_vectors, word\_pairs)

if not similarities:

return render\_template("index.html", error="No similarities calculated.")

# Form top 8 unique pairs

top\_8\_pairs = form\_8\_unique\_pairs(similarities)

# Prepare similarity matrix for grouping

similarity\_matrix = np.zeros((len(filtered\_words), len(filtered\_words)))

for i, word1 in enumerate(filtered\_words):

for j, word2 in enumerate(filtered\_words):

similarity\_matrix[i, j] = np.dot(

word\_vectors[word1], word\_vectors[word2]

) / (np.linalg.norm(word\_vectors[word1]) \* np.linalg.norm(word\_vectors[word2]))

# Group word pairs into groups

grouped\_pairs = group\_word\_pairs(top\_8\_pairs, similarity\_matrix, filtered\_words)

# Compute success rate based on at least two matching words

success\_rate = calculate\_success\_rate(original\_groups, grouped\_pairs)

# Generate interactive PCA plot for word embeddings

pca\_plot = generate\_pca\_plot(model, filtered\_words, grouped\_pairs)

return render\_template("index.html", words=words, original\_groups=original\_groups,

generated\_groups=grouped\_pairs, success\_rate=success\_rate, pca\_plot=pca\_plot)

return render\_template("index.html")

if \_\_name\_\_ == "\_\_main\_\_":

app.run(debug=True)

**output**

A screenshot of a computer

Description automatically generated

**Web Interface: Word Group Generator**

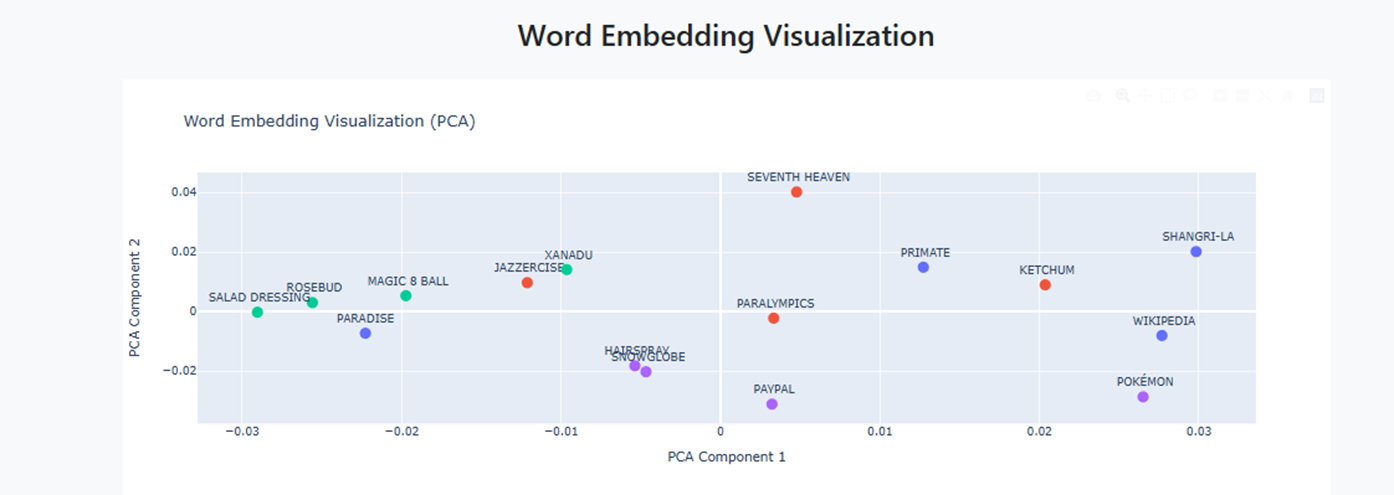
**A screenshot of a computer

Description automatically generated**

**A screenshot of a group

Description automatically generated**

**Word Embedding Visualization**

****

**Future  
Recommendations**

* + Refine the group generation process to ensure the formation of all correct groups level by level.
  + Use iterative approach to enhance the model's overall efficiency and reliability.
  + Train the model with a larger dataset containing more puzzles to improve its understanding of semantic relationships and grouping accuracy.